Using THINK Rez

27

Resource description files are the text files that THINK Rez compiles to produce resources for your application.

If you are not familiar with Apple's Resource description language, Rez, we recommend that you use a graphical resource editor, such as ResEdit (included in your package) or Resorcerer. See Apple Computer's *ResEdit 2.1 Reference* (Addison-Wesley) for instructions on using ResEdit.

This chapter describes how to use THINK Rez to compile resource description files and add resources to your projects. It describes how to write a resource description file, how to set your THINK Rez options for your project, and also how THINK Rez is different from Rez, the resource compiler that comes with Apple's Macintosh Programmer's Workshop.

Contents

The Resource Compiler 531
- Using a resource compiler. 531
- Standard type declaration files 532

Structure of a Resource Description File 532
- Sample resource description file. 533

Resource Description Statements 534
- Syntax notation 534
- Special terms 535
- Data—specify raw data. 535
- Type—declare resource type 536
- Resource—specify resource data 547

Labels 551
- Built-in functions to access resource data 551
- Declaring labels within arrays 552
- Label limitations 553
- Using labels: two examples 554

Preprocessor Directives 557
Variable definitions 558
Header file processing 559
If-then-else processing 560

Resource Description Syntax 561
Numbers and literals 561
Expressions 562
Variables and functions 563
Strings 565

Setting THINK Rez Options 568
Resource alignment 568
Redeclared types ok 568
Prefix String 569

Differences from MPW Rez 569

The Resource Compiler

The resource compiler, THINK Rez, compiles a text file (or files) called a resource description file and adds the resources to your project resource file. THINK Rez has preprocessor directives that let you substitute macros, include other files, and use if-then-else constructs. (These directives are described in the section "Preprocessor Directives" later in this chapter.)

Using a resource compiler

THINK Rez creates new resources in your project resource file. Use it with the Resource Copier, which copies resources that already exist into your project resource file, to add resources to your applications. Figure 27-1 illustrates the process of creating a project resource file.

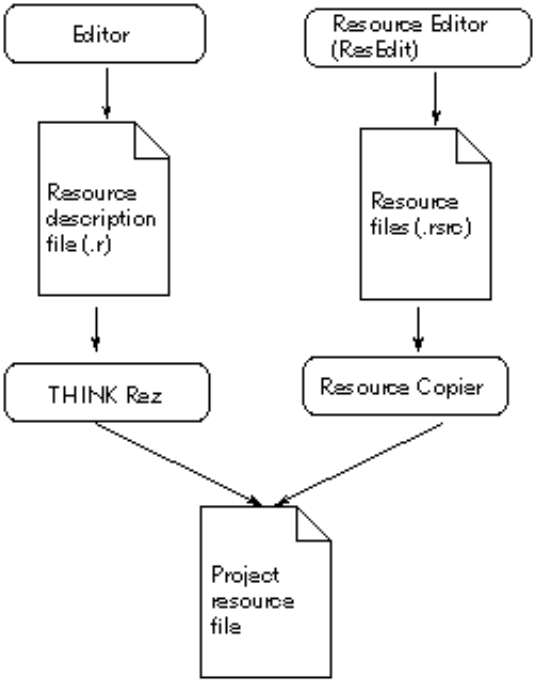

Figure 27-1 Creating a resource file

Standard type declaration files

Types.r, SysTypes.r, and Pict.r are examples of text files containing resource declarations for standard resource types. These files are located in the Rez #Includes folder.

This file…	Contains…
Types.r	Type declarations for the most common Macintosh resource types ('ALRT', 'DITL', 'MENU', and so on).
SysTypes.r	Type declarations for 'DRVR', 'FOND', 'FONT', 'FWID', 'INTL', 'NFMT', and many others.
Pict.r	Type declaration for PICT resources for debugging PICTs.

Table 27-1 Examples of standard type declaration files

Structure of a Resource Description File

A resource description file consists of resource type declarations (which can be included from another file) followed by resource data for the declared types. Note that the resource compiler and resource decompiler have no built-in resource types. You need to define your own types or include the appropriate .r files.

A resource description file may contain any number of these statements:

Statement	Description
data	Specify raw data
type	Type declaration — declare resource type descriptions for subsequent resource statements
resource	Data specification — specify data for a resource type declared in a previous type statement

Table 27-2 Statements in a resource description file

Each of these statements is described in the sections that follow.

A type declaration describes how the declaration of a resource of that type will look. You must declare a type (with a type

statement) before you make a resource of that type (with a `resource` statement). Otherwise, you can freely mix `type` and `resource` statements in a file. You can redefine a type any number of times, even a type defined in `Types.r`, `SysTypes.r`, or `Pict.r`.

A resource description file can also include comments and preprocessor directives. Comments can be included any place white space is allowed in a resource description file, by putting them within the comment delimiters `/*` and `*/`. Note that comments do not nest. For example, this is *one* comment:

```
/* Hello /* there */
```

THINK Rez also supports C++ style comments:

```
type 'tost' {  // Ignore rest of this line.
```

Preprocessor directives substitute macro definitions and `include` files, and provide if-then-else processing before other THINK Rez processing takes place. The syntax of the preprocessor is very similar to that of the C language preprocessor.

Sample resource description file

An easy way to learn about the resource description format is to decompile some existing resources. For example, using SADeRez to decompile an application's resources might generate this:

```
resource 'WIND' (128, "Sample Window") {
   {64, 60, 314, 460},
   documentProc,
   visible,
   noGoAway,
   0x0,
   "Sample Window"
};
```

This resource data corresponds to the following type declaration, contained in `Types.r`:

```
type 'WIND'{
   rect;                    /* bounds    */
   integer documentProc,    /* procID    */
      altDBoxProc, plainDBox,
      altDBoxProc, noGrowDocProc,
      movableDBoxProc, zoomDocProc = 8,
      zoomNoGrow=12, rDocProc=16;
   byte invisible, visible;/* visible   */
   fill byte;
   byte noGoAway, goAway;  /* close box */
   fill byte;
   unsigned hex longint;   /* refCon    */
   pstring Untitled="Untitled";
                           /* title     */
   /* . . . */
}
```

Type and resource statements are explained in detail in the reference section that follows.

Resource Description Statements

This section describes the syntax and use of the three types of resource description statements available for the resource compiler: `data`, `type`, and `resource`.

Syntax notation

The following syntax notation is used with resource description statements:

Notation	Description
`terminal`	Plain text indicates a word that must appear in the statement exactly as shown. Special symbols (such as, `-`, `*`, =) and punctuation (such as “`,`” and “`;`”) must also be entered exactly as shown.
nonterminal	Items in italics can be replaced by anything that matches their definition.
`[` *optional* `]`	Square brackets mean that the enclosed elements are optional.

Table 27-3 Syntax notation

Notation	Description
repeated …	An ellipsis (…), when it appears in the text of this reference only, indicates that the preceding item can be repeated one or more times.
a | *b*	A vertical bar indicates an either/or choice.
(*grouping*)	Parentheses indicate grouping (useful with the | and … notation).
'[' *x* ']'	Curly single quotation marks ('…') indicate that one of the syntax notation characters (for example, [or]) must be written as a literal. In this example, the brackets would be typed literally. They do *not* mean that the *x* is optional.

Table 27-3 Syntax notation *(Continued)*

Spaces between syntax elements, constants, and punctuation are optional. They are used for readability only.

Tokens in resource description statements may be separated by spaces, tabs, returns, or comments.

Special terms

The following terms represent a minimal subset of the nonterminal symbols used to describe the syntax of commands in the resource description language:

Expression is defined in the section "Expressions" later in this chapter.

Term	Definition
resource-type	long-expression
resource-name	string
resource-ID	word-expression
ID-range	*ID* [: *ID*]

Table 27-4 Special terms

A full syntax definition can be found at the end of this chapter.

Data—specify raw data

Use the `data` statement to specify raw data as a sequence of bits, without any formatting.

Syntax

```
data res-type '(' ID [,resource-name] [,attributes...] ')' '{'
   data-string
'}' ;
```

Description

A data statement reads the data found in *data-string* and writes it as a resource with the type *res-type* and the ID *ID*. You can optionally specify a resource name, resource attributes, or both.

For example,

```
data 'PICT' (128) {
   $"4F35FF8790000000"
   $"FF234F35FF790000"
};
```

Type—declare resource type

A type declaration provides a template that defines the structure of the resource data for a single resource type or for individual resources. If more than one type declaration is given for a resource type, the last one read before the data definition is the one that's used. This lets you override declarations from include files or previous resource description files.

Syntax

```
type resource-type [ '(' ID-range ')' ] '{'
   type-specification...
'}' ;
```

Description

A type declaration causes any subsequent `resource` statement for the type *resource-type* to use the declaration {*type-specification*...}. The optional *ID-range* specification causes the declaration to apply only to a given resource ID or range of IDs.

Type-specification is one of the following:

Specification	Description
`bitstring[`*n*`]`	Bitstring of n bits
`byte`	8-bit number
`integer`	16-bit number
`longint`	32-bit number
`Boolean`	1-bit Boolean value
`char`	8-bit character
`string`	Plain (no length indicator or termination string character)
`pstring`	Pascal string
`wstring`	Word string
`cstring`	C string
`point`	Point
`rect`	Rect
`fill`	Zero fill
`align`	Zero fill to nibble, byte, word, or long word boundary
`switch`	Control construct (case statement)
`array`	Array data specification—zero or more instances of data types

Table 27-5 Type specifications for type statement

These types can be used singly or together in a `type` statement. Each of these type specifiers is described in the sections that follow.

Note

Several of these types require additional fields. The exact syntax is given in the sections that follow.

You can also declare a resource type that uses another resource's type declaration by using the following variant of the type statement:

```
type  resource-type1  [ '(' ID-range ')' ]
   as resource-type2  [ '(' ID ')' ] ;
```

Data-type specifications

A data-type statement declares a field of the given data type. It can also associate symbolic names or constant values with the data type.

The data-type specification can take three forms, as shown in this example:

```
type 'XAMP' {
   /* declare a resource of type 'XAMP' */
   byte;
   byte off=0, on=1;
   byte = 2;
};
```

The first `byte` statement declares a byte field; the actual data is supplied in a subsequent `resource` statement.

The second `byte` statement is identical to the first, except that the two symbolic names `off` and `on` are associated with the values `0` and `1`. These symbolic names could be used in the `resource` data.

The third `byte` statement declares a byte field whose value is always `2`. In this case, no corresponding statement would appear in the `resource` data.

Numeric expressions and strings can appear in `type` statements; they are defined in the section "Expressions" later in this chapter.

Numeric types. The numeric types (`bitstring`, `byte`, `integer`, `longint`) are fully specified like this:

```
[ unsigned ]  [ radix ]  numeric-type
   [ = expr | symbol-definition… ];
```

The `unsigned` prefix signals a resource decompiler, such as MPW DeRez, that the number should be displayed without a sign—that the high-order bit can be used for data and the value of the integer cannot be negative. The `unsigned` prefix is ignored by THINK Rez but is needed by a resource decompiler to correctly represent a decompiled number. THINK Rez uses a sign if it is specified in the data. Precede a signed negative constant with a minus sign (`–`); 0x`FFFFFF85` and `–0x7B` are equivalent in value.

Radix is one of the following string constants:

```
hex            decimal        octal
binary         literal
```

You can supply numeric data as decimal, octal, hexadecimal, binary, or literal data.

Numeric-type is one of the following:

Type	Description
`bitstring` '`[`'*length*'`]`'	Declare a bitstring of *length* bits (maximum 32)
`byte`	Declare a byte (8-bit) field (this is the same as `bitstring[8]`)
`integer`	Integer (16-bit) field (this is the same as `bitstring[16]`)
`longint`	Long integer (32-bit) field (this is the same as `bitstring[32]`)

Table 27-6 Numeric type specifiers

THINK Rez uses integer arithmetic and stores numeric values as integer numbers. THINK Rez translates `boolean`, `byte`, `integer`, and `longint` values to `bitstring` equivalents. All computations are done in 32 bits and truncated.

An error is generated if a value won't fit in the number of bits defined for the type. The valid ranges for values of `byte`, `integer`, and `longint` constants are as follows:

Type	Maximum	Minimum
`byte`	`255`	`–128`
`integer`	`65,535`	`–32,768`
`longint`	`4,294,967,295`	`–2,147,483,648`

Table 27-7 Value ranges for numeric type specifiers

Boolean type. A `Boolean` is a single bit with two possible states: `0` (or `false`) and `1` (or `true`) (`true` and `false` are global predefined identifiers.) `boolean` values are declared as follows:

```
boolean [ = constant | symbolic-value... ];
```

The type `boolean` declares a 1-bit field; this is equivalent to:

```
unsigned bitstring[1]
```

Note

This type is not the same as a `boolean` variable as defined by Pascal.

Character type. Characters are declared as follows:

```
char  [ = string | symbolic-value... ];
```

Type `char` declares an 8-bit field (this is the same as writing `string[1]`).

Here is an example:

```
type 'SYMB'  {
   char  dollar = "$", percent = "%";
};

resource 'SYMB' (128)  {
   dollar
};
```

String type. String data types are specified like this:

string-type `[`‘`[`’ *length* ‘`]`’`]` `[=` *string* `|` *symbol-value...*`];`

String-type is one of the following:

String type	Description
`[hex] string`	Plain string has no length indicator or termination character generated. The optional `hex` prefix tells a resource decompiler, like MPW DeRez, to display it as a hex string. `string[`*n*`]` contains *n* characters and is *n* bytes long. The type `char` is shorthand for `string[1]`.
`pstring`	Pascal string has a leading byte containing the length information. `pstring[`*n*`]` contains *n* characters and is *n*+1 bytes long. `pstring` has a built-in maximum length of 255 characters, the highest value the length byte can hold. If the string is too long to fit the field, a warning is given and the string is truncated.

Table 27-8 String type specifiers

String type	Description
`wstring`	Word string is a very large `pstring`. Its length is stored in the first two bytes. Therefore, a word string can contain up to 65,535 characters. `wstring[`*n*`]` contains n characters and is *n*+2 bytes long.
`cstring`	C string generates a trailing null byte. `cstring[`*n*`]` contains *n*–1 characters and is *n* bytes long. A C string of length 1 can be assigned only the value `""`, because `cstring[1]` has room only for the terminating null.

Table 27-8 String type specifiers *(Continued)*

Each string type may be followed by an optional *length* indicator in brackets (`[`*n*`]`). *Length* is an expression indicating the string length in bytes. *Length* is a positive number in the range `1` ≤ *length* ≤ `2,147,483,647` for `string` and `cstring`, `1` ≤ *length* ≤ `255` for `pstring`, and `1` ≤ *length* ≤ `65,535` for `wstring`.

Note

You cannot assign the value of a literal to a string type.

If no length indicator is given, a `pstring`, `wstring`, or `cstring` stores the number of characters in the corresponding data definition. If a length indicator is given, the data may be truncated on the right or padded on the right. The padding characters for all string types are nulls. If the data contains more characters than the length indicator provides for, the string is truncated and a warning message is given.

Warning

A null byte within a `cstring` is a termination indicator and may confuse C programs and a resource decompiler like MPW DeRez. However, the full string, including the explicit null and any text that follows it, will be stored by THINK Rez as input.

Point and rectangle types. Because points and rectangles appear so frequently in resource files, they have their own simplified syntax:

```
point [ = point-constant | symbolic-value... ];
rect [ = rect-constant | symbolic-value... ];
```

where

point-constant = '{' *x-integer-expr*, *y-integer-expr* '}'

and

rect-constant = '{' *integer-expr*, *integer-expr*,
integer-expr, *integer-expr* '}'

These `type` statements declare a point (two 16-bit signed integers) or a rectangle (four 16-bit signed integers). The integers in a rectangle definition specify the rectangle's upper-left and lower-right points, respectively.

Fill and align types

The resource created by a resource definition has no implicit alignment. It's treated as a bit stream, and integers and strings can start at any bit. The `fill` and `align` type specifiers are two ways of padding fields so that they begin on a boundary that corresponds to the field type. `align` is automatic and `fill` is explicit. Both `fill` and `align` generate zero-filled fields.

Fill specification. The `fill` statement causes THINK Rez to add the specified number of bits to the data stream. The fill is always `0`. The form of the statement is

```
fill fill-size [ '[' length ']' ] ;
```

where *fill-size* is one of the following strings:

```
bit             nibble          byte
word            long
```

These declare a fill of `1`, `4`, `8`, `16`, or `32` bits (optionally multiplied by the *length* modifier). *Length* is an expression ≤ `2,147,483,647`.

The following fill statements are equivalent:

```
fill word[2];
fill long;
fill bit[32];
```

The full form of a `type` statement specifying a fill might be:

```
type 'XRES' {
  data-type specifications;
  fill bit[2];
};
```

Note

THINK Rez supplies zeros as specified by `fill` and `align` statements.

Align specification. Alignment causes THINK Rez to add fill bits of zero value until the data is aligned at the specified boundary. An alignment statement takes the following form:

```
align  align-size ;
```

where *align-size* is one of these strings:

```
nibble        byte          word
long
```

Alignment pads with zeros until data is aligned on a 4-, 8-, 16-, or 32-bit boundary. This alignment affects all data from the point where it is specified until the next `align` statement.

Array type

An array is declared as follows:

```
[wide] array [ array-name | '[' length ']' ]  '{'
  array-list
'}';
```

The *array-list*, a list of type specifications, is repeated zero or more times. The wide option outputs the array data in a wide display format (in a resource decompiler like MPW DeRez)—the elements that make up the array-list are separated by a comma and space instead of a comma, Return, and Tab. Either *array-name* or [*length*] may be specified. *Array-name* is an identifier.

If the array is named, then a preceding statement should refer to that array in a constant expression with the `$$CountOf(`*array-name*`)`

function; otherwise, a resource decompiler like MPW DeRez will treat the array as open-ended. For example,

```
type 'STR#' {
   /* define a string list resource */
   integer = $$Countof(StringArray);
   array StringArray {
      pstring;
   };
};
```

The `$$CountOf()` function returns the number of array elements (in this case, the number of strings) from the resource data.

If [*length*] is specified, there must be exactly *length* elements.

Array elements are generated by commas. Commas are element separators. Semicolons are element terminators. In this example, however, it may be a good idea to use semicolons as element separators:

```
type 'xyzy' {
   array Increment {
      integer = $$ArrayIndex(Increment);
   };
};

resource 'xyzy' (0) {
   {  /* zero elements */
   }
};

resource 'xyzy' (1) {
   {  /* two elements */
      ,
   }
};

resource 'xyzy' (3) {
   }  /* two elements */
      ;;
   }
};

/* The only way to specify one element in
 * an array that has all constant elements,
 * is to use a semicolon terminator.
 */
```

```
resource 'xyzy' (4) {
   {  /* one element */
      ;
   }
};
```

Switch type

The `switch` statement specifies a number of case statements for a given field or fields in the resource. The format is:

`switch` '{' *case-statement...* '}';

where a *case-statement* has this form:

`case` *case-name* `:` `[` *case-body* `;` `]`...

Case-name is an identifier. *Case-body* may contain any number of type specifications and must include a single constant declaration per case, in this form:

`key` *data-type* `=` *constant*

The case which applies is based on the key value. For example,

```
type 'DITL'{
   /* dialog item list declaration
    * from Types.r
    */

   ...type specifications...

   switch {    /* one of the following */
   case Button:
      boolean enabled, disabled;
      key bitstring[7] = 4; /* key value */
      pstring;
   case CheckBox:
      boolean enabled, disabled;
      key bitstring[7] = 5; /* key value */
      pstring;

   ...and so on.

   };
};
```

Sample type statement

The following sample `type` statement is the standard declaration for a `'WIND'` resource, taken from the `Types.r` file:

```
type 'WIND'{
   rect;                    /* bounds    */
   integer documentProc,    /* procID    */
      altDBoxProc, plainDBox,
      altDBoxProc, noGrowDocProc,
      movableDBoxProc, zoomDocProc = 8,
      zoomNoGrow=12, rDocProc=16;
   byte invisible, visible;/* visible   */
   fill byte;
   byte noGoAway, goAway;  /* close box */
   fill byte;
   unsigned hex longint;   /* refCon    */
   pstring Untitled="Untitled";
                           /* title     */
   /* . . . */
}
```

The type declaration consists of header information followed by a series of statements, each terminated by a semicolon (`;`). The header of the sample window declaration is

```
type 'WIND'
```

The header begins with the `type` keyword followed by the name of the resource type being declared—in this case, a window. You may specify a standard Macintosh resource type, as shown in *Inside Macintosh VI*, Chapter 13, "Resource Manager," or you may declare a resource type specific to your application.

The left brace (`{`) introduces the body of the declaration. The declaration continues for as many lines as necessary until a matching right brace (`}`) is encountered. You can write more than one statement on a line, and a statement may cover more than one line (like the `integer` statement above). Each statement represents a field in the resource data. Recall that comments may appear wherever white space appears in the resource description file; comments begin with `/*` and end with `*/`, as in C.

Symbol definitions

Symbolic names for data type fields simplify the reading and writing of resource definitions. Symbol definitions have the form

`name =` *value* `[,` *name* `=` *value* `]...`

For numeric data, the "= *value*" part of the statement can be omitted. If a sequence of values consists of consecutive numbers, the explicit assignment can be left out—if *value* is omitted, it's assumed to be one greater than the previous value. (The value is assumed to be zero if it's the first value in the list.) This is true for `bitstring`s and their derivatives: `byte`, `integer`, and `longint`. For example,

```
integer  documentProc, dBoxProc, plainDBox,
         altDBoxProc, noGrowDocProc,
         zoomProc=8, rDocProc=16;
```

In this example, the symbolic names `documentProc`, `dBoxProc`, `plainDBox`, `altDBoxProc`, and `noGrowDocProc` are automatically assigned the numeric values `0`, `1`, `2`, `3`, and `4`.

Memory is the only limit to the number of symbolic values that can be declared for a single field. There is also no limit to the number of names you can assign to a given value. For example:

```
integer  documentProc=0, dBoxProc=1,
         plainDBox=2, altDBoxProc=3,
         rDocProc=16, Document=0, Dialog=1,
         DialogNoShadow=2,
         ModelessDialog=3,
         DeskAccessory=16;
```

Resource—specify resource data

`Resource` statements specify actual resources, based on previous type declarations.

Syntax

`resource` *res-type* '`(`' *ID* [`,`*resource-name*] [`,`*attributes*] '`)`' '`{`'
    [*data-statement* [`,`*data-statement*]...]
'`}`';

Description

A `resource` statement specifies the data for a resource of type *res-type* and ID *ID*. The latest type declaration declared for *res-type* is used to parse the data specification. *Data-statements* specify the actual data; *data-statements* appropriate to each resource type are defined in the next section.

The resource definition causes an actual resource to be generated. A `resource` statement can appear anywhere in the resource description file, or even in a separate file specified on the command

line or as an `#include` file, as long as it comes after the relevant type declaration.

Data statements
The body of the data specification contains one data statement for each declaration in the corresponding type declaration. The base type must match the declaration.

Base type	Instance types
`string`	`string`, `cstring`, `pstring`, `wstring`, `char`
`bitstring`	`boolean`, `byte`, `integer`, `longint`, `bitstring`
`rect`	`rect`
`point`	`point`

Table 27-9 Data statement specifications

Switch data. Switch data statements are specified by using this format:

```
switch-name   data-body
```

For example, the following could be specified for the `'DITL'` type given earlier:

```
...
CheckBox { enabled, "Check here" },
...
```

Array data. Array data statements have this format:

‘{’ [*array-element* [, *array-element*]…] ‘}’

where an *array-element* consists of any number of data statements separated by commas.

For example, the following data might be given for the `'STR#'` resource defined earlier:

```
resource 'STR#' (280) {
   {  "this",
      "is",
      "a",
      "test"
   }
};
```

Sample resource definition

See Inside Macintosh: Macintosh Toolbox Essentials, *"Window Manager," for information about resources in windows.*

This section describes a sample resource description file for a window. Here, again, the type declaration is given in the section "Sample type statement" earlier in this chapter.

```
type 'WIND'{
   rect;                    /* bounds    */
   integer documentProc,    /* procID    */
      altDBoxProc, plainDBox,
      altDBoxProc, noGrowDocProc,
      movableDBoxProc, zoomDocProc = 8,
      zoomNoGrow=12, rDocProc=16;
   byte invisible, visible;/* visible   */
   fill byte;
   byte noGoAway, goAway;  /* close box */
   fill byte;
   unsigned hex longint;   /* refCon    */
   pstring Untitled="Untitled";
                           /* title     */
   /* . . . */
}
```

Here is a typical example of the window data corresponding to this declaration:

```
resource 'WIND' (128, "My window", appheap,
                 preload) {
                   /* Status window      */
   {40,80,120,300},/* Bounding rectangle */
   documentProc,   /* documentProc etc.. */
   Visible,        /* Visible or         */
                   /*   Invisible        */
   goAway,         /* GoAway or NoGoAway */
   0,              /* Reference value    */
                   /*   RefCon           */
   "Status Report" /* Title              */
   };
```

This data definition declares a resource of type `'WIND'`, using whatever type declaration was previously specified for `'WIND'`. The resource ID is `128`; the resource name is `"My window"`. Because the resource name is represented by the Resource Manager as a `pstring`, it should not contain more than 255 characters. The resource name may contain any character, including the null character (`'\000'`). The resource will be placed in the application heap when loaded, and it will be loaded when the resource file is opened.

The first statement in the window type declaration declares a bounding rectangle for the window:

```
rect;
```

The rectangle is described by two points: the upper-left corner and the lower-right corner. The points of a rectangle are separated by commas like this:

```
{ top, left, bottom, right }
```

An example of data for these coordinates is

```
{ 40, 80, 120, 300 }
```

Symbolic names. Symbolic names may be associated with particular values of a numeric type. Notice that a symbolic name is given for the data in the second, third, and fourth fields of the window declaration. For example,

```
integer  documentProc=0, dBoxProc=1,
         plainDBox=2, altDBoxProc=3,
         noGrowDocProc=4, zoomProc=8,
         rDocProc=16;      /* windowType */
```

This statement specifies a signed 16-bit integer field with symbolic names associated with the values 0 to 4, 8, and 16. The values 0 through 4 need not be indicated in this case; if no values are given, symbolic names are automatically given values starting at 0, as explained previously.

In the sample window declaration, the values `True` (`1`) and `False` (`0`) were given to two different byte variables. For clarity, those symbolic names were used in the window's resource data. That is:

```
visible,
goAway,
```

were used instead of their equivalents:

```
TRUE,
TRUE,
```

or

```
1,
1,
```

Labels

Labels support some of the more complicated resources, such as `'NFNT'` and color QuickDraw resources. Use labels within a resource type declaration to calculate offsets and permit accessing of data at the labels.

Syntax

label ::= *character* {*alphanum*}... ':'
character ::= '_' | A | B | C …
number ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
alphanum ::= *character* | *number*

Description

Labeled statements are valid only within a resource type declaration. Labels are local to each type declaration. More than one label can appear in a statement.

Labels may be used in expressions. In expressions, use only the identifier portion of the label (that is, everything up to, but excluding, the colon). See the section "Declaring labels within arrays" on the next page for more information.

The value of a label is always the offset, *in bits*, between the beginning of the resource and the position at which the label occurs when mapped to the resource data. In this example,

```
type 'cool' {
    cstring;
endOfString:
    integer = endOfString;
};

resource 'cool' (8) {
  "Neato"
}
```

the integer following the `cstring` would contain:

```
( len("Neato") [5] + null byte [1] ) *
8 [bits per byte]
   = 48.
```

Built-in functions to access resource data

In some cases, accessing the actual resource data to which a label points is desirable. Several built-in functions allow access to that data:

`$$BitField(`*label*`, `*startingPosition*`, `*numberOfBits*`)`
Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label*.

`$$Byte(`*label*`)`
Returns the byte found at *label*.

`$$Word(`*label*`)`
Returns the word found at *label*.

`$$Long(`*label*`)`
Returns the longword found at *label*.

For example, the resource type `'STR '` could be redefined without using a `pstring`. Here is the definition of `'STR '` from `Types.r`:

```
type 'STR ' {
   pstring;
}
```

Here is a redefinition of `'STR '` using labels:

```
type 'STR ' {
len:  byte = (stop - len) / 8 - 1;
      string[$$Byte(len)];
stop:;
};
```

Declaring labels within arrays

Labels declared within arrays may have many values. For every element in the array, there is a corresponding value for each label defined within the array. Use array subscripts to access the individual values of these labels. The subscript values range from 1 to *n* where *n* is the number of elements in the array. Labels within arrays that are nested in other arrays require multidimensional subscripts. Each level of nesting adds another subscript. The rightmost subscript varies most quickly. Here is an example:

```
type 'test' {
   integer = $$CountOf(array1);
   array array1 {
      integer = $$CountOf(array2);
      array array2 {
foo:     integer;
      };
   };
};
```

```
resource 'test' (128) {
   {
      {1,2,3},
      {4,5}
   }
};
```

In the above example, the label `foo` takes on values listed below:

```
foo[1,1] = 32     $$Word(foo[1,1]) = 1
foo[1,2] = 48     $$Word(foo[1,2]) = 2
foo[1,3] = 64     $$Word(foo[1,3]) = 3
foo[2,1] = 96     $$Word(foo[2,1]) = 4
foo[2,2] = 112    $$Word(foo[2,2]) = 5
```

This built-in function may be helpful in using labels within arrays:

`$$ArrayIndex(`*arrayname*`)`
This function returns the current array index of the array *arrayname*. An error occurs if this function is used anywhere outside the scope of the array *arrayname*.

Label limitations

Keep in mind that THINK Rez is basically a one pass compiler. This characteristic explains some of the limitations of labels.

Note

To decompile a given type, that type must not contain any expressions with more than one undefined label. An undefined label is a label that occurs lexically after the expression. To define a label, use it in an expression before the label is defined.

This example demonstrates how expressions can contain only one undefined label:

```
type 'test' {
   /* In the expression below,
    * start is defined,
    * next is undefined.
    */
start:   integer = next - start;
   /* In the expression below,
    * next is defined because it was
    * used in a previous expression,
    * but final is undefined.
    */
middle:  integer = final - next;
next:    integer;
final:
};
```

Actually, THINK Rez can compile types that have expressions containing more than one undefined label, but a resource decompiler like DeRez cannot decompile those resources. It simply generates data resource statements.

Note

The label specified in `$$BitField()`, `$$Byte()`, `$$Word()`, and `$$Long()` must occur lexically before the expression; otherwise, an error is generated.

Using labels: two examples

The first example shows the `'ppat'` declaration using labels. Without using labels, the whole end section of the resource would have to be combined into a single hex string (everything following the `PixelData` label). Using labels, the complete `'ppat'` definition can be expressed in THINK Rez language.

```
/* PixPat record */

type 'ppat' {
   integer oldPattern, newPattern, ditherPattern;
                     /* Pattern type           */
   unsigned longint = PixMap / 8;
                     /* Offset to pixmap       */
   unsigned longint = PixelData / 8;
                     /* Offset to data         */
   fill long;        /* Expanded pixel image   */
```

```
   fill word;        /* Pattern valid flag     */
   fill long;        /* expanded pattern       */
   hex string [8];   /* old-style pattern      */


/* PixMap record */
PixMap:
   fill long;        /* Base address           */
   unsigned bitstring[1] = 1;
                     /* New pixMap flag        */
   unsigned bitstring[2] = 0;
                     /* Must be 0              */
   unsigned bitstring[13];
                     /* Offset to next row     */
   rect;             /* Bitmap bounds          */
   integer;          /* pixMap vers number     */
   integer unpacked; /* Packing format         */
   unsigned longint; /* size of pixel data     */
   unsigned hex longint;
                     /* horizontal resolution  */
                     /* (ppi) (fixed)          */
   unsigned hex longint;
                     /* vertical resolution    */
                     /* (ppi) (fixed)          */
   integer chunky, chunkyPlanar, planar;
                     /* Pixel storage format   */
   integer;          /* # bits in pixel        */
   integer;          /* # bits per field       */
   integer;          /* # components in pixel  */
   unsigned longint; /* Offset to next plane   */
   unsigned longint = ColorTable / 8;
                     /* Offset to color table  */
   fill long;        /* Reserved               */

PixelData:
   hex string [(ColorTable - PixelData) / 8];

   ColorTable:
      unsigned hex longint;
                        /* ctSeed           */
      integer;          /* transIndex       */
      integer = $$Countof(ColorSpec) - 1;
                        /* ctSize           */
      wide array ColorSpec {
         integer;             /* value      */
         unsigned integer;    /* RGB: red   */
         unsigned integer;    /* green      */
         unsigned integer;    /* blue       */
      };
   };
```

Here is another example of a resource definition that uses labels. In this example, the `$$BitField()` function is used to access information stored in the resource, in order to calculate the size of the various data areas added at the end of the resource. Without labels, all data would have to be combined into one hex string.

```
/* IconPMap (pixMap) record */

type 'cicn' {
   fill long;        /* Base address           */
   unsigned bitstring[1] = 1;
                     /* New pixMap flag        */
   unsigned bitstring[2] = 0;
                     /* Must be 0              */
pMapRowBytes: unsigned bitstring[13];
                     /* Offset to next row     */
Bounds: rect;        /* Bitmap bounds          */
   integer;          /* pixMap vers number     */
   integer unpacked; /* Packing format         */
   unsigned longint; /* Size of pixel data     */
   unsigned hex longint;
                     /* horizontal resolution  */
                     /* (ppi) (fixed)          */
   unsigned hex longint;
                     /* vertical resolution    */
                     /* (ppi) (fixed)          */
   integer chunky, chunkyPlanar, planar;
                     /* Pixel storage format   */
   integer;          /* # bits in pixel        */
   integer;          /* # components in pixel  */
   integer;          /* # bits per field       */
   unsigned longint; /* Offset to next plane   */
   unsigned longint; /* Offset to color table  */
   fill long;        /* Reserved               */

   /* IconMask (bitMap) record */
   fill long;        /* Base address           */
maskRowBytes: integer;
                     /* Row bytes              */
   rect;             /* Bitmapds               */

   /* IconBMap (bitMap) record */
   fill long;        /* Base address           */
iconBMapRowBytes: integer;
                     /* Row bytes              */
   rect;             /* Bitmap bounds          */
   fill long;        /* Handle placeholder     */
```

```
    /* Mask data */
    hex string [$$Word(maskRowBytes) *
       ( $$BitField(Bounds, 32, 16)  /* bottom */
         - $$BitField(Bounds, 0, 16) /* top    */
       )];

    /* BitMap data */
    hex string [$$Word(iconBMapRowBytes) *
       ($$BitField(Bounds, 32, 16)   /* bottom */
         - $$BitField(Bounds, 0, 16) /* top    */
       )];

    /* Color Table */
    unsigned hex longint;   /* ctSeed           */
    integer;                /* transIndex       */
    integer = $$Countof(ColorSpec) - 1;
                            /* ctSize           */
    wide array ColorSpec {
       integer;             /* value            */
       unsigned integer;    /* RGB: red         */
       unsigned integer;    /* green            */
       unsigned integer;    /* blue             */
    };

    /* PixelMap data */
    hex string [$$BitField(pMapRowBytes,0,13) *
        ($$BitField(Bounds,32,16)    /* bottom */
          - $$BitField(Bounds, 0, 16)/* top    */
        )];
};
```

Preprocessor Directives

Preprocessor directives substitute macro definitions and include files and provide if-then-else processing before other THINK Rez processing takes place.

The syntax of the preprocessor is very similar to that of the C language preprocessor. Preprocessor directives must observe these rules and restrictions:

1. Each preprocessor statement must be expressed on a single line, beginning on a new line and terminated by a return character.

2. The pound sign (#) must be the first character on the line of the preprocessor statement (except for spaces and tabs).

3. Identifiers (used in macro names) may be letters (A-Z, a-z), digits (0–9), or the underscore character (_).

4. Identifiers may be any length.

5. Identifiers may not start with a digit.

6. Identifiers are not case sensitive.

Variable definitions

The #define and #undef directives let you assign values to identifiers:

```
#define  macro  data
#undef   macro
```

The #define directive causes any occurrence of the identifier *macro* to be replaced with the text *data*. You can extend a macro over several lines by ending the line with the backslash character (\), which functions as the THINK Rez escape character. For example,

```
#define poem "I wander \
thro\' each \
charter\'d street"
```

(Quotation marks within strings must also be escaped.)

#undef removes the previously defined identifier *macro*. You can also define and undefine macros in the Prefix String section of the **THINK Rez options** dialog, described later in this chapter. The following macros are defined in the Prefix String section of the THINK Rez options dialog.

Variable	Value	Description
true	1	Use for the Boolean value true
false	0	Use for the Boolean value false
rez	1	Use to test whether a resource compiler is running
derez	0	Use to test whether a resource de-compiler is running
THINK_Rez	1	Use to test which resource compiler is running

Table 27-10 Predefined macros

Header file processing

You can include the text of another file in your file with the `#include` directive. It works like the `#include` directive in C:

```
#include file
```

Include the text file *file*. The maximum nesting is to ten levels. For example,

```
#include "MyTypes.r"
```

You can enclose the file name in either quotes (`""`) or angle brackets (`<>`). Table 27-11 shows the rules THINK Rez uses to find header files.

#include statement	THINK Rez...
`<filename.h>`	Looks only in the THINK tree
`"filename.h"`	Looks first in the referencing folder, then in the project tree, and finally in the THINK tree.

Table 27-11 Header file search rules

The referencing folder is the one that contains the file which has the `#include` preprocessor directive. For example, if a source file references a header file `MyUtils.h` and that file in turn has the line `#include "MyUtilTypes.h"`, THINK Rez will look for `MyUtilTypes.h` in the folder that contains `MyUtils.h` first.

Apple's MPW Rez does not support angle brackets in the `#include` directive. Instead, you enclose all file names in quotes. If you plan to use your code with both THINK Rez and Apple's MPW Rez, test for which compiler is running with the `THINK_Rez` macro, like this:

```
#ifdef THINK_Rez
#include <filename.h>
#else
#include "filename.h"
#endif
```

Pragma once directives

You may want to create a header file that you want included in several places but which should define its symbols only once in a file. You can use the `#pragma once` directive to do this.

If you have the directive:

```
#pragma once
```

in your header file, THINK Rez will include that file only once. If another file tries to include that header file, THINK Rez knows that the symbols in that file have already been defined, so it doesn't process the file again.

Keep these restrictions in mind when you use `#pragma once`:

- It's case sensitive. For example, if you include a file once with the statement `#include <myres.h>` and later with the statement `#include <MyRes.h>`, THINK Rez will include it twice.
- It doesn't distinguish between files included with `<...>` and `"..."`. For example, suppose you have two header files named `xyz.h`; one is in the THINK tree and the other is in the project tree. If you include one with `#include <xyz.h>` and another with `#include "xyz.h"`, THINK Rez will not include the second file.
- It ignores characters after the first 32. If you include two files with names that start with the same 32 characters, THINK Rez will not include the second file.

If-then-else processing

These directives provide conditional processing.

```
#if expression
[  #elif expression ]
[  #else ]
#endif
```

Expression is defined in the section "Expressions" later in this chapter. When used with the `#if` and `#elif` directives, *expression* may also include this expression:

`defined` *identifier*

or

`defined` '`(`' *identifier* '`)`'

The following may also be used in place of `#if`:

```
#ifdef macro
#ifndef macro
```

For example,

```
#define Thai
Resource  'STR '  (199)  {
#ifdef English
   "Hello"
#elif defined (French)
   "Bonjour"
#elif defined (Thai)
   "Sawati"
#elif defined (Japanese)
   "Konnichiwa"
#endif
};
```

Resource Description Syntax

This section describes the details of the resource description syntax.

Numbers and literals

All arithmetic is performed as 32-bit signed arithmetic. These are the basic constants:

- Decimal (*nnn*…): Signed decimal constant between 4,294,967,295 and -2,147,483,648.
- Hex (`0X`*hhh*… or `$`*hhh*…): Signed hexadecimal constant between `0X7FFFFFFF` and `0X80000000`.
- Octal (`0`*ooo*…): Signed octal constant between `017777777777` and `020000000000`.
- Binary (`0B`*bbb*…): Signed binary constant between `0B11111111111111111111111111111111` and `0B10000000000000000000000000000000`.
- Literal (`'`*aaaa*`'`): A literal may contain one to four characters. Characters are printable ASCII characters or escape characters. If there are less than four characters in the literal, then the characters to the left (high bits) are assumed to be null characters (`'\000'`). Characters that are not in the printable character set (and are not the characters `\'` and `\\`, which have special meanings) can be escaped ac-

cording to the character escape rules. (See the section "Strings" later in this chapter.)

Literals and numbers are treated in the same way by the resource compiler. A literal is a value within single quotation marks. For instance, `'A'` is a number with the value 65; on the other hand, `"A"` is the character A expressed as a string. Both are represented in memory by the bitstring `01000001`. (Note, however, that `"A"` is not a valid number and `'A'` is not a valid string.) The following numeric expressions are all equivalent:

```
'B'
66
'A'+1
```

Literals are padded with nulls on the left side so that the literal `'ABC'` is stored as shown in Figure 27-2.

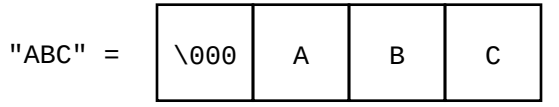

Figure 27-2 Padding of literals

Expressions

An expression may consist of simply a number or literal. Expressions may also include numeric variables, labels, and system functions.

This table lists the operators in order of precedence with highest precedence first. Groupings indicate equal precedence. Evaluation is always left to right when the priority is the same. Variables are defined following the table.

Operator	Meaning
(*expr*)	Use to force precedence in expressions
-expr	Arithmetic (two's complement) negation of *expr*
~expr	Bitwise (one's complement) negation of *expr*
!expr	Logical negation of *expr*
*expr1 * expr2*	Multiplication
expr1 / expr2	Division

Table 27-12 Operator precedence

Operator	Meaning
expr1 % *expr2*	Remainder from dividing *expr1* by *expr2*
expr1 + *expr2*	Addition
expr1 - *expr2*	Subtraction
expr1 << *expr2*	Shift left—shift *expr1* left by *expr2* bits
expr1 >> *expr2*	Shift right—shift *expr1* right by *expr2* bits
expr1 > *expr2*	Greater than
expr1 >= *expr2*	Greater than or equal to
expr1 < *expr2*	Less than
expr1 <= *expr2*	Less than or equal to
expr1 == *expr2*	Equal
expr1 != *expr2*	Not equal
expr1 & *expr2*	Bitwise AND
expr1 ^ *expr2*	Bitwise XOR
expr1 | *expr2*	Bitwise OR
expr1 && *expr2*	Logical AND
expr1 || *expr2*	Logical OR

Table 27-12 Operator precedence *(Continued)*

The logical operators `!`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `&&`, and `||` evaluate to `1` (true) or `0` (false).

Variables and functions

Some resource compiler variables contain commonly used values. All THINK Rez variables start with `$$` followed by an alphanumeric identifier.

The following variables and functions have string values (typical values are given in parentheses):

`$$Date`
Current date. Useful for putting timestamps into the resource file. The format is generated through the ROM call `IUDateString()`. (`"Thursday, May 20, 1987"`)

`$$Format("`*formatString*`", `*arguments*`)`
Works just like the `#printf` directive except that `$$format()` returns a string rather than printing to standard output.

`$$Name`
Name of resource from the current resource. The current resource is the resource being generated in a `resource` statement, or being included from an `include` statement.

For example, to include all `'DRVR'` resources from one file and keep the same information, but also set the SYSHEAP attribute:

```
INCLUDE "file" 'DRVR' (0:40) AS
 'DRVR' ($$ID, $$Name, $$Attributes | 64);
```

The `$$Type`, `$$ID`, `$$Name`, and `$$Attributes` variables are undefined outside of a `change`, `delete`, `include`, or `resource` statement.

`$$Time`
Current time. Useful for time-stamping the resource file. The format is generated through the ROM call `IUTimeString()`.
(`"7:50:54 AM"`)

`$$Version`
Version number of THINK Rez. (`"V3.0"`)

These variables and functions have numeric values:

`$$Attributes`
Attributes of resource from the current resource. See the `$$Name` string variable.

`$$BitField(`*label*`, `*startingPosition*`, `*numberOfBits*`)`
Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label*.

`$$Byte(`*label*`)`
Returns the byte found at *label*.

`$$Day`
Current day. Range 1–31.

`$$Hour`
Current hour. Range 0–23.

`$$ID`
ID of resource from the current resource. See the `$$Name` string variable.

`$$Long(`*label*`)`
Returns the longword found at *label*.

`$$Minute`
Current minute. Range 0–59.

`$$Month`
Current month. Range 1–12.

`$$PackedSize(`*Start*`, `*RowBytes*`, `*RowCount*`)`
Given an offset (*Start*) into the current resource and two integers, *RowBytes* and *RowCount*, this function calls the Toolbox routine `UnpackBits()` *RowCount* times. `$$PackedSize()` returns the unpacked size of the data found at *start*. Use this function only for decompiling resource files. An example of this function is found in `Pict.r`.

`$$ResourceSize`
Current size of resource in bytes. When decompiling, `$$ResourceSize` is the actual size of the resource being decompiled. When compiling, `$$ResourceSize` returns the number of bytes that have been compiled so far for the current resource. (See the `'KCHR'` resource in `SysTypes.r` for an example.)

`$$Second`
Current second. Range 0–59.

`$$Type`
Type of resource from the current resource. See the `$$Name` string variable.

`$$Weekday`
Current day of the week. Range 1–7 (that is, Sunday–Saturday).

`$$Word(`*label*`)`
Returns the word found at *label*.

`$$Year`
Current year.

Strings

There are two basic types of strings:

- Text string ("*a*…"): The string can contain any printable character except ‘ " ’ and ‘\’. These and other characters can be created through escape sequences. (See the section “Escape characters” on page 567 the next page.) The string "" is a valid string of length 0.

- Hex string ($"*hh*…"): Spaces and tabs inside a hexadecimal string are ignored. There must be an even number of hexadecimal digits. The string $"" is a valid hexadecimal string of length 0.

Any two strings (hexadecimal or text) will be concatenated if they are placed next to each other with only white space in between. (In this case, returns and comments are considered white space.)

Figure 27-3 shows a Pascal string declared as

```
pstring [10];
```

whose data definition is

```
"Hello".
```

\005	H	e	l	l	o	\000	\000	\000	\000	\000

Figure 27-3 Internal representation of a Pascal string

In the input file, string data is surrounded by double quotation marks ("). You can continue a string on the next line. A separating token (for example, a comma or brace) signifies the end of the string data. A side effect of string continuation is that a sequence of two quotation marks ("") is simply ignored. For example,

```
"Hello ""out "
"there."
```

is the same string as:

```
"Hello out there.";
```

To place a quotation mark character within a string, precede the quotation mark with a backslash like this:

```
"Hello \"out\" there."
```

Escape characters

The backslash character (\) is provided as an escape character to allow you to insert nonprintable characters in a string. For example, to include a newline character in a string, use the escape sequence `\n`.

These are the valid escape sequences:

Name	Escape Sequence	Hex Value
Tab	`\t`	`$09`
Backspace	`\b`	`$08`
Return	`\r`	`$0A`
Newline	`\n`	`$0D`
Form feed	`\f`	`$0C`
Vertical tab	`\v`	`$0B`
Rubout	`\?`	`$7F`
Backslash	`\\`	`$5C`
Single quotation mark	`\'`	`$3A`
Double quotation mark	`\"`	`$22`

Table 27-13 Escape sequences

You can also use octal, hexadecimal, decimal, and binary escape sequences to specify characters that do not have predefined escape equivalents. The forms are:

Base	Form (# digits)	Example
2	`\0B`*bbbbbbbb* (8)	`\0B01000001`
8	`\`*ooo* (3)	`\101`
10	`\0D`*ddd* (3)	`\0D065`
16	`\0X`*hh* (2)	`\0X41`
16	`\$`*hh* (2)	`\$41`

Table 27-14 Numeric escape sequences

Here are some examples:

```
\077         /* 3 octal digits          */
\0xFF        /* '0x' + 2 hex digits     */
\$F1\$F2\$F3 /* '$' + 2 hex digits      */
\0d099       /* '0d' + 3 decimal digits */
```

Note

An octal escape code consists of exactly three digits. For instance, to place an octal escape code with a value of 7 in the middle of an alphabetic string, write `AB\007CD`, not `AB\7CD`.

Setting THINK Rez Options

To set options for THINK Rez, choose **THINK Rez** from the hierarchical menu that appears when you choose **Options** from the **Edit** menu.

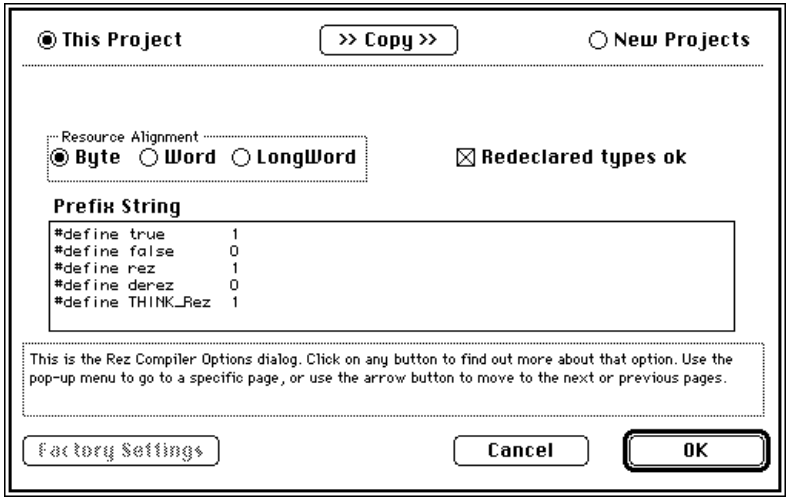

Figure 27-4 The THINK Rez options dialog

Resource alignment

This option lets you choose how your resources are aligned: along byte, word, or longword boundaries. The default is byte.

Redeclared types ok

This option lets you choose if THINK Rez continues compiling your file if you declare a resource type more than once in a resource file. If it's selected, THINK Rez prints a warning and continues compiling. Otherwise, THINK Rez prints an error message and stops compiling. The option is selected by default.

Prefix String

The THINK Rez Prefix String lets you include some text in all the THINK Rez source files in your project. The effect is the same as if you put the string at the beginning of all your resource description files. By default, the Prefix String defines the macro `true` to be `1`, `false` to be `0`, `rez` to be `1`, `derez` to be `0`, and `THINK_Rez` to be `1`.

If you need to define a macro in all your files, define it here. For example, you may have some resources you use for debugging that are compiled only if the macro `DEBUG` is defined. To include that code, include the following line:

```
#define DEBUG
```

When you no longer need to include the debugging code, just delete that line from this page. You don't need to edit every `.r` file in your project.

Note

When you change the THINK Rez prefix string, the THINK Project Manager recompiles all the THINK Rez files in your project.

Differences from MPW Rez

THINK Rez is based on Rez, Apple's resource utility that's available with Apple's Macintosh Programmer's Workshop (MPW). THINK Rez supports most of the Rez features described in the MPW 3.2 documentation.

The most significant difference between MPW Rez and THINK Rez is that some of MPW Rez's features are handled by the THINK Project Manager and not by THINK Rez. THINK Rez only compiles resource description files; MPW Rez also modifies resource files and copies them into your application. The THINK Project Manager handles these tasks. To copy a resource file into your application, simply include it in your project and the THINK Project Manager copies its contents into your application automatically. To modify a resource file, double-click its name in your project window and the THINK Project Manager launches your resource editor (such as ResEdit or Resourcerer) and opens the file.

The following are *not* supported in THINK Rez:

- The `change` and `delete` statements
- The `$$shell` function

The following is a feature in THINK Rez that is not in MPW Rez.

In THINK Rez, the `#include` statement uses the same rules for finding header files that THINK C and Symantec C++ use. For more information, see the section “Header file processing” earlier in this chapter.